Monitorização da performance de linguagens de programação

Daniel Du
Departamento Informática
Universidade do Minho
Braga, Portugal
PG53751@alunos.uminho.pt

Ricardo Lucena
Departamento Informática
Universidado do Minho
Braga, Portugal
PG54187@alunos.uminho.pt

José Fonte
Departamento Informática
Universidado do Minho
Braga, Portugal
a91775@alunos.uminho.pt

***Abstract*— O presente relatório detalha um estudo realizado no âmbito da Unidade Curricular de Experimentação em Engenharia de Software sobre o consumo energético e o desempenho de benchmarks desenvolvidos em diferentes linguagens programação executados com e sem a aplicação de limites de potência, conhecidos como powercaps. A análise foi realizada com foco na relação entre o consumo de energia, o tempo de execução e os powercaps aplicados. Observamos que os powercaps têm um impacto significativo no consumo de energia e no tempo de execução dos programas, com variações notáveis entre as diferentes linguagens de programação. Os resultados revelaram que linguagens de baixo nível, como Haskell, tendem a ter um consumo de energia menor e uma eficiência de execução mais alta, enquanto linguagens de alto nível, como Python, mostram um consumo de energia mais elevado. Este estudo destaca a importância da consideração da eficiência energética na programação e no desenvolvimento de software, bem como a necessidade de estratégias para uma computação mais verde e sustentável.**

***Index terms*—Powercap, Eficiência energética, Haskell, Java, Python, Tempo de execução, Consumo energético, Desempenho**

I. Introdução

Ao longo das últimas décadas, a computação tornou-se cada vez mais presente no nosso quotidiano, impulsionando um grande avanço tecnológico e a inovação em diversas áreas. Contudo, este crescimento exponencial também trouxe consigo um aumento significativo no consumo de energia, contribuindo para preocupações ambientais e questões relacionadas com a eficiência energética.

Neste relatório, apresentaremos os resultados do nosso estudo que se centra na análise comparativa do consumo energético de benchmarks, desenvolvidos em diferentes linguagens de progração, executados com e sem a aplicação de powercaps. Através da recolha e análise de dados rigorosos, procuramos não só compreender o impacto das restrições energéticas no consumo total de energia, como também avaliar como essas restrições afetam o desempenho dos programas em termos de tempo de execução dos benchmarks.

Utilizando uma variedade de técnicas estatísticas e metodologias de análise de dados procuramos identificar padrões, tendências e correlações significativas que possam auxiliar o engenheiro de software a escolher a linguagem de programação mais adequada para as suas necessidades, tendo em conta a eficiência energética.

II. Trabalhos relacionados

Com o aumento da consciencialização ambiental e o crescimento da recente industrialização informática, isto é, grandes centros de dados e computação surgem agora preocupações relativas a *clean IT* e como o atingir.

A eficência e consumo de energia perante as várias linguagens de programação é uma das ramificações da nova preocupação e tem sido amplamente estudado. Um estudo notável na àrea é um estudo realizado por Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes e João Saraiva denominado por *"Energy Efficiency across Programing Languages"* aonde exploram a relação entre eficiência energética, tempo e memória em 27 diferentes linguagens de programação. Os resultados mostram como linguagens mais rápidas ou mais lentas influênciam o consumo de mais ou menos energia e como o uso de memória ajuda os engenheiros de software a decidir qual linguagem usar quando a eficiência energética é uma preocupação. [1]

III. Metodologia

A. Benchmarks

Para a realização deste estudo recomemos a três linguagens de programação - *Haskell*, *Java* e *Python*, com os seguintes benchmarks *Nofib, Dacapo* e *pyperformance*, respetivamente.

1) ***Nofib**:* Em relação ao Nofib, os benchmarks corridos eram todos single threaded e estão divididos em quatro categorias: *Imaginary*, *Spectral*, *Real* e *Shootout*.

Os benchmarks na categoria *Imaginary* são benchmarks mais dedicados à solução de puzzles, como por exmeplo o *n-queens*. Na categoria *Spectral* são benchmarks direcionados à algoritmia do kernel. Os do *Real* são benchmarks de aplicações reais com interface de linha de comando. Por último os que estão na categoria *Shootout* são mais direcionados a benchmarks de jogos. [2]

2) ***Dacapo**:* No que toca ao benchmark *Dacapo*, este apresenta uma enorme variedade de subcategorias de benchmarks e nos executamos 10 delas , sendo estas as seguintes: *avrora, batik, biojava, eclipse, kafka, spring, tomcat, graphchi, jme e jythom*.

Os testes no *Avrora* simula um número de programas que correm numa grelha de microcontroladores AV. No *batik* é produzido um número de imagens Scalable Vector Graphics (SVG) baseadas em testes unitários em Apache Batik. Na pilha de testes do *Eclipse* são executados testes para o Eclipse IDE. [3] O *Kafka* é uma plataforma de streaming de eventos distribuídos, *open source*, usada por milhares de empresas para *pipeline* de dados, análises de streaming, integração de dados de alta performance. [4] O *Biojava* é uma livraria open-source dedicada ao processamento de dados biológicos, como por exemplo, análise de rotinas estáticas, parsing de ficheiros e manipulação de sequências biológicas e estruturas 3D. [5] O *Tomcat* executa um conjunto de *queries* num servidor Tomcat de forma a recuperar e verificar as páginas web resultantes. [3] O *GraphChi* consegue correr longas computações gráficas em apenas uma máquina, através de um algoritmo para processamento de gráficos do disco (SSD ou HDD). [6] O *jme* é um moderno motor de jogos. O *Spring* é uma [7] Por último, o *jython* interpretador do python escrito em java. [8], [3]

3) ***Pyperformance**:* Para o benchmark *pyperformance* também corremos dez sub-benchmarks, sendo estes os seguintes: *Chamelean, Docutils, Html5Lib, 2to3, Tornado_http, Nbody, Json_dumps, Pidigits, Async_tree e Django_template*.

No *chameleon* é renderizado um *template* usando o modulo chameleon para criar uma table HTML com 500 linhas e 10 colunas. No *docutil*, é convertido um documento do tipo Docutil para o tipo HTML. No *html5lib* é dado o *parse* de um ficheiro HTML usando a livraria html5lib. No *2to3* não só mede a performance do python em si, como também mede a performance do modulo lib2to3, que por sua vez pode variar consoante a versão do python. No *tornado_http*, é testado se o número de conexões é escalável a dez mil de conexões abertas. [9] O *nbody* é um benchmark proveniente do *Computer Language Benchmarks Game*. O *Json_dumps* são funções do modulo json. No *pidigits* são calculados dois mil digitos do π (pi). No *async_tree* trabalho assíncrono, na qual chama o método *asyncio.gather()* numa àrvore de seis níveis de profundidade com seis ramificações por nível, os nodos folha similuam o trabalho assíncrono. Por último, o *django_template* usa o sistema *Django template* para construír uma tabela HTMl com cento e cinquenta linhas e colunas, totalizando 22500 células. [10]

B. Medições de energia

Para o estudo do consumo de energia de cada programa consoante o PowerCap imposto recomemos à utilização do RAPL.

1) *Rapl:* O RAPL é um projeto que fornece uma interface desenvolvida em C para a gestão dos *Intel Running Average Power Limit*. Dado o uso desta ferramenta para a geração dos resultados obtidos face ao powercap imposto era imperativo o uso de computadores com processadores da *Intel* que suportassem *RAPL - Sandy Bridge*, ou seja, processadores a partir da *2ª geração Intel Core* até aos mais recentes até à data.

O Intel RAPL permite o software alterar o power cap em componentes *hardware* como por exemplo o processador ou a memória principal. Os componentes são capazes de se gerirem de forma a respeitarem o power cap imposto enquanto tenta otimizar a performance. É de notar que power cap e consumo de energia não são a mesma coisa, sendo que o primeiro apenas especifica uma barreira superior no consumo de energia. [11]

2) *Ambiente de trabalho, PowerLimits e Calibração do Rapl:* Para a execução do RAPL com os benchmarks, uma vez que tinhamos na posse dois computadores com processador Intel, decidimos então executa-los em ambas as máquinas de forma a também perceber se a diferença de computador iria ter ou não impacto na performance e consumo de energia dos benchmarks. Na tabela abaixo estão apresentadas as características de ambos os computadores.

	Computador 1	Computador 2
Modelo	Lenovo IdeaPad 5 14IIL05	Asus X550JX 1.0
SO	Kubuntu 22.04.4 LTS x86_64	Linux Mint 21 x86_64
[Processdor] Modelo	Intel i7-1065G7	Intel i7-4720HQ
[Processdor] Cores	4	4
[Processdor] Threads	8	8
[Processdor] Freq. Base	1.3 GHz	1.6 GHz
[Processdor] Freq Max	3,9 GHz	3.6 GHz

	Computador 1	Computador 2
[Processdor] Cache	8 MB	6 MB
[Processdor] Sup. Mem	DDR4-3200	DDR3L 1333/1600
RAM	16 GB	8 GB
GPU1	Intel Iris Plus Graphics G7	Intel HD Graphics 4600
GPU2	NVIDIA GeForce MX350	NVIDIA GeForce GTX 950M

Dado que o processador presente, segundo a Intel, no *computador1* trabalha entre 15W e 25W, [12] e no *computador2* entre 12W e 47W [13] nós decidimos então usar os seguintes powercaps: −1, 12, 15, 25, 47 e 55.

O uso do valor −1 foi usado para medições dos benchmarks sem powercap imposto, o valor 12 é quando estamos a dar ao processador a maior limitação energética no computador2, ou seja, quando estamos a dar apenas 12W (Watts), o valor 15 é quando estamos a dar ao processador a maior limitação energética no computador1, ou seja, quando estamos a dar apenas 15W (Watts), o valor 25 é quando estamos a dar ao processador a menor limitação energética no computador1, ou seja, estamos a dar 25W (Watts), o valor 47 é quando estamos a dar ao processador a menor limitação energética no computador2, ou seja, quando estamos a dar 47W (Watts), por último, o valor 55 é quando estamos a dar ao processador menos limitação energética do que a menor limitação energética que ambos os processadores precisam.

Após termos corrido em ambos os computadores o RAPL com os benchmarks duas vezes mais dois programas de menor dimensão (*Fibonacci* em *c e c++*), verificamos que os valores obtidos nos nossos computadores eram quer no RAPL com os benchmarks quer nos programas de menor dimensão, dúbios. Posto isto, mais várias discussões com a equipa docente foi-nos concedida a permissão de usar os valores obtidos de um outro grupo. Estes resultados dúbios serão apresentados na secção "IV - Resultados".

Sendo assim, as características do computador do outro grupo eram as seguintes:

	Valores
Modelo	Asus Vivobook X1505ZA
SO	Ubuntu 22.04.4 LTS x86_64
[Processdor] Modelo	Intel i7-12700H
[Processdor] Total Cores	14 (6 performance, 8 efficient)
[Processdor] Threads	20
[Processdor] Freq. Base	2.3 GHz
[Processdor] Freq Max	4,7 GHz
[Processdor] Cache	24 MB
[Processdor] Sup. Mem	DDR5 4800, DDR4 3200, LPDDR5 5200, LPDDR4x 4267
RAM	16 GB
GPU1	Intel Alder Lake-P

Dado que o processador presente, segundo a Intel, no *computador* trabalha entre 15W e 45W, [14] foi então usado os seguintes powercaps: −1, 5, 10, 35, 45 e 115.

O uso do valor −1, como nos outros computadores, foi usado para medições dos benchmarks sem powercap imposto, o valor 5 é quando estamos a dar ao processador a maior limitação energética, inclusive mais do que a maior limitação tabelada, o valor 10 é quando estamos a dar ao processador a maior limitação energética no processador segundo os valores tabelados da Intel, ou seja, quando estamos a dar apenas 10W (Watts), o valor 35 é quando estamos a dar ao processador um valor intermédio de energia ao processador, o valor 45 é quando estamos a dar ao processador a menor limitação energética ao processador segundo os valores tabelados da Intel, ou seja, quando estamos a dar 47W (Watts), por último, o valor 115 é quando estamos a dar ao processador menos limitação energética do que a menor limitação energética que o processador necessita.

3) *Modo de Execução:*

Com o programa pronto a ser executado, selecionamos 10 *benchmarks* de cada linguagem, sem critério de seleção em concreto. Assim, corremos isoladamente os *benchmarks* para cada linguagem, gravando os resultados num único ficheiro. Cada *benchmark* foi executado 10 vezes por cada nível de *PowerLimit*, o que deu origem a ficheiros com 600 linhas de data para analisar.

IV. Resultados

A. Análise dos primeiros resultados dúbios obtidos

Em primeiro lugar fizemos um teste com o programa *Fibonacci* dado juntamente com o *RAPL*. Os valores obtidos podem ser visualizados no seguinte gráfico:

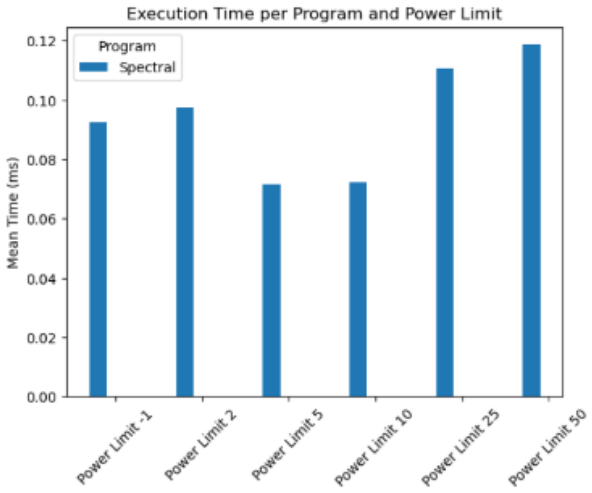

Figure 1: Histograma sobre a execução da função *Fibonacci* sobre diferentes *PowerCaps*

Como podemos ver por esta imagem, os valores obtidos não fazem qualquer tipo de sentido. Seria de esperar um decréscimo quase constante do tempo de execução, com o valor de 2 no *PowerLimit* a ser o que demora mais tempo, e −1 ou 50 o mais rápido. No entanto vemos que há uma descida na transição de 2 para 5 Watts, e uma subida na transição de 10 para 25, algo que o grupo não consegue justificar.

Independentemente de obtermos resultados negativos, o grupo decidiu correr os *benchmarks* para ver se iríamos obter resultados maus. Após o programa ter executado, reparamos que os valores pareciam ser negativos, mas decidimos analisá-los melhor para confirmar.

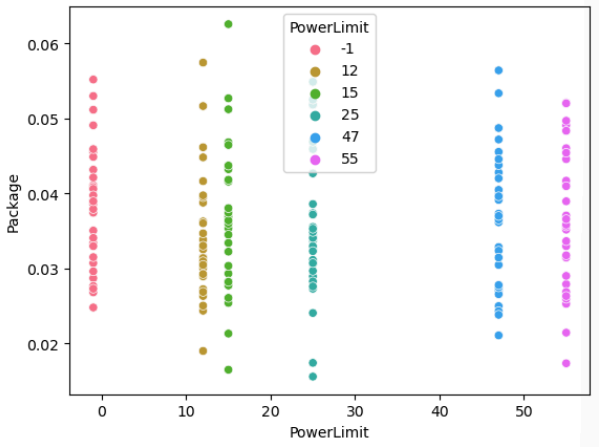

Figure 2: *Scatter Plot* para todos os *benchmarks* de *haskell* a comparar os valores *Package* e *PowerLimit*

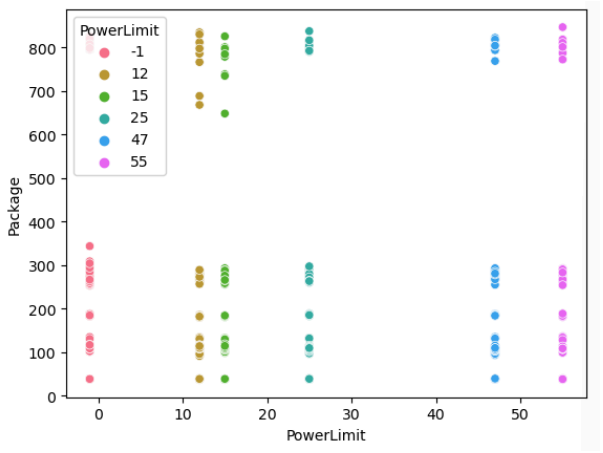

Figure 3: *Scatter Plot* para todos os *benchmarks* de *java* a comparar os valores *Package* e *PowerLimit*

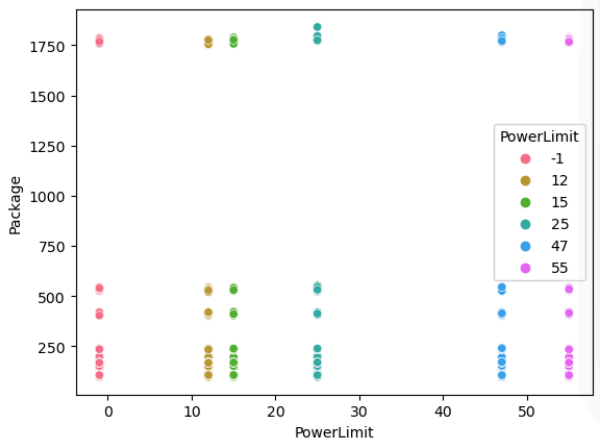

Figure 4: *Scatter Plot* para todos os *benchmarks* de *python* a comparar os valores *Package* e *PowerLimit*

Como podemos observar pelos gráficos acima representados não há quase qualquer diferença entre a aplicação de *PowerLimit* aos programas. Mas como estes gráficos apenas nos dão uma visão geral de todos os *benchmarks* das linguagens, e não para cada *benchmark* isoladamente, decidimos calcular a correlação entre as variáveis *PowerLimit* e *Package* para cada *benchmark*.

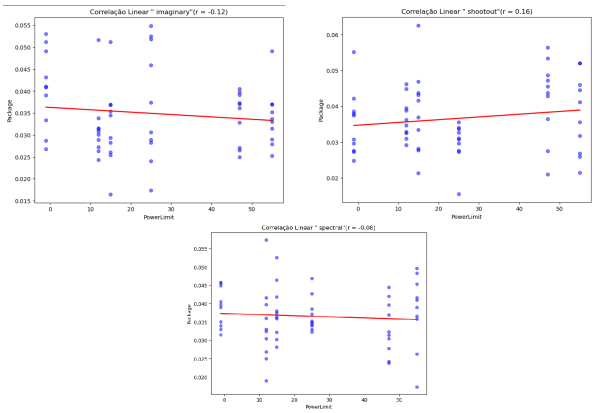

Figure 5: Correlação entre a variável *PowerLimit* e *Package* de alguns *benchmarks* de *haskell*.

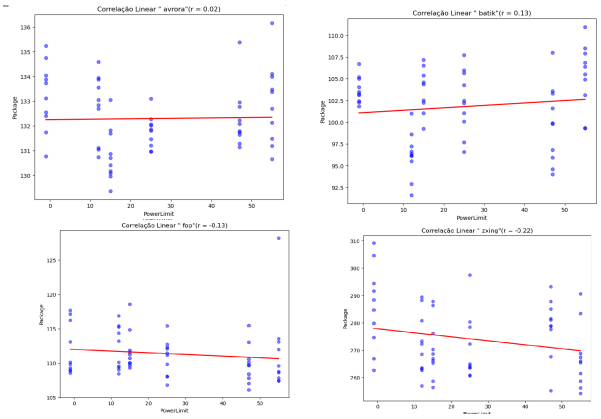

Figure 6: Correlação entre a variável *PowerLimit* e *Package* de alguns *benchmarks* de *java*.

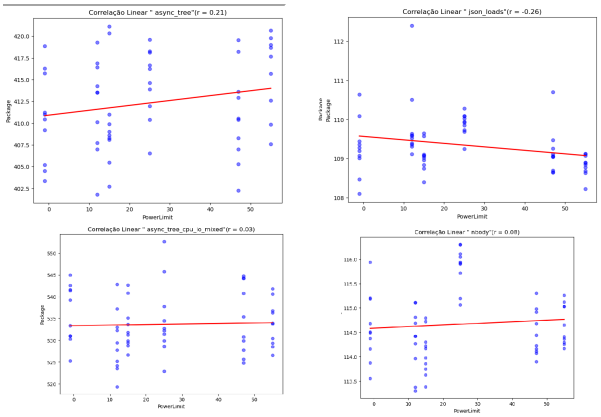

Figure 7: Correlação entre a variável *PowerLimit* e *Package* de alguns *benchmarks* de *python*.

Como podemos observar, após obter a regressão linear, as retas que obtemos não são consistentes ou seja, quer obtemos correlações positivas, quer correlações negativas. Para além disso calculamos a correlação de *Pearson* para todos os *benchmarks*. Esta relação dita que se o valor calculado de "p" for inferior a 0.05, a correlação entre as duas variáveis testadas é estatisticamente significativa. Para praticamente todos os *benchmarks* o valor de "p" era superior a 0.05, o que prova que não há evidências suficientes para provar que *Package* e *PowerLimit* são estatisticamente significantes.

Como os nossos resultados não permitem que seja feita uma análise em detalhe sobre a aplicação do *PowerCap* aos diversos *benchmarks* avançamos com o estudo da data de outro grupo, como explicamos anteriormente.

B. Correlação entre o PowerCap e o consumo de energia

Depois de obtermos os dados decidimos analisar todas as linguagens através de um gráfico de clustering.

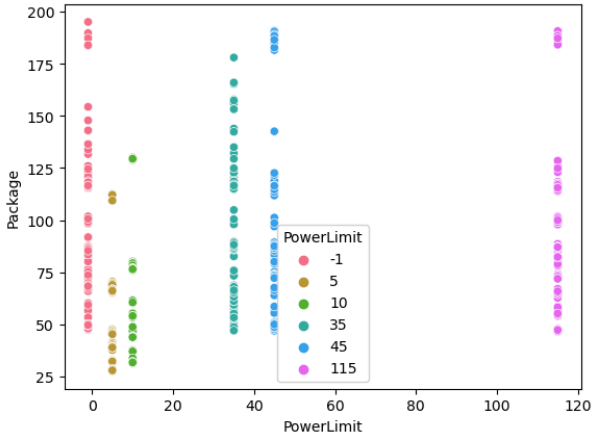

Figure 8: *Scatter Plot* para todos os *benchmarks* de *haskell* a comparar os valores *Package* e *PowerLimit*

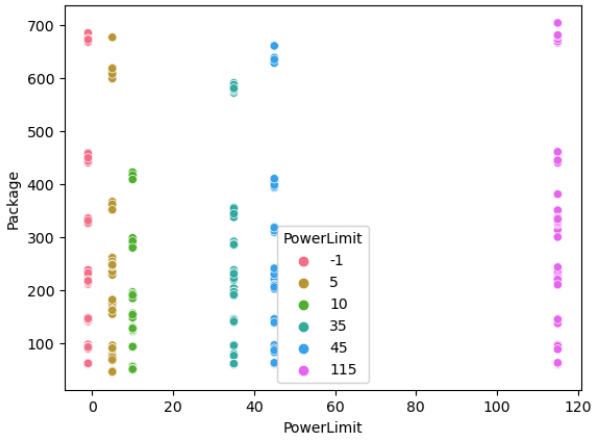

Figure 9: *Scatter Plot* para todos os *benchmarks* de *java* a comparar os valores *Package* e *PowerLimit*

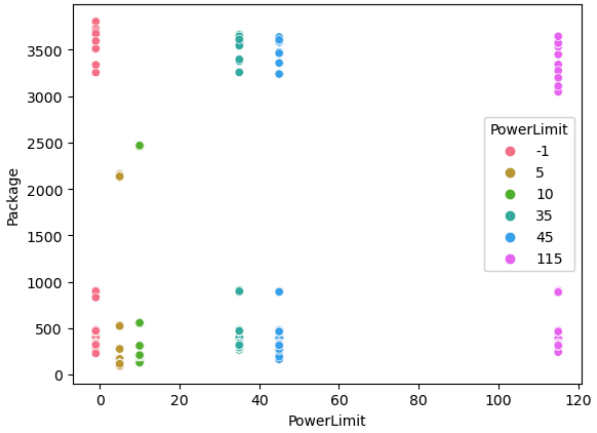

Figure 10: *Scatter Plot* para todos os *benchmarks* de *python* a comparar os valores *Package* e *PowerLimit*

Após visualização dos gráficos de cluster podemos concluir que quando o *PowerLimit* tem valores mais baixos, nomeadamente 5 e 10, o *Package* tende a diminuir. No entanto, este diminuição estagna a partir dos valor de 35 no *PowerLimit* onde a partir daí todos os valores de *Package* tendem a ser iguais. Para ter uma maior perceção dos valores que temos, calculados um *box plot* para cada *benchmark*:

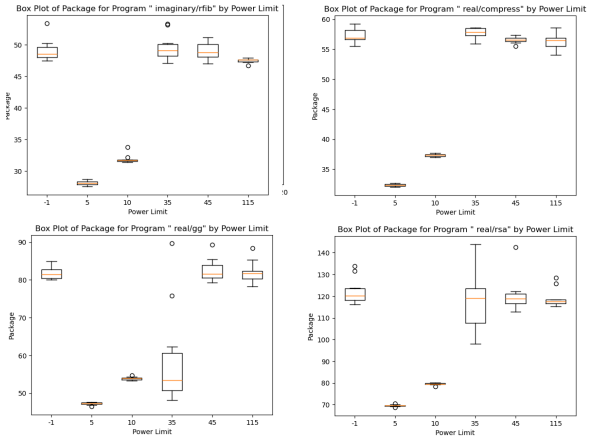

Figure 11: *Box Plot* para *benchmarks* de *haskell* a comparar os valores *Package* e *PowerLimit*

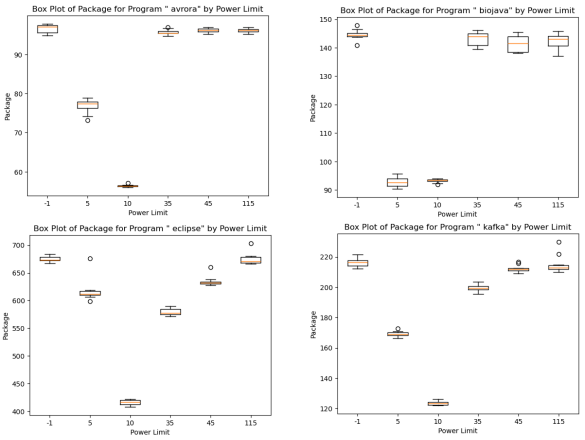

Figure 12: *Box Plot* para *benchmarks* de *java* a comparar os valores *Package* e *PowerLimit*

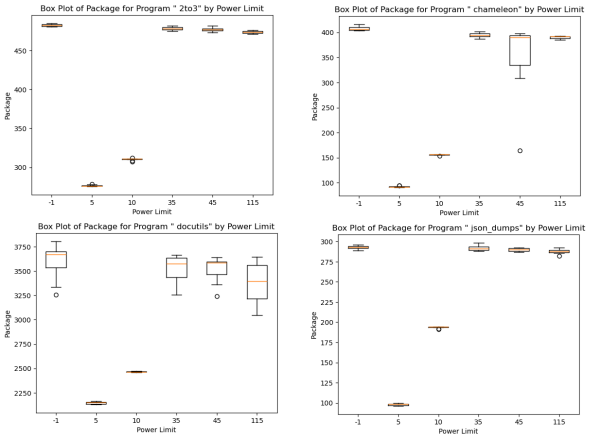

Figure 13: *Box Plot* para *benchmarks* de *python* a comparar os valores *Package* e *PowerLimit*

Mais uma vez, conseguimos analisar de forma mais precisa a estagnação dos valores do *Package* a partir de valores de *PowerLimit* 35, indicando que no RAPL do ambiente de teste só vale a pena limitar os Watts para 35, no máximo. Algo curioso que é possível verificar é que em maior parte dos *benchmarks* de Java, o *Package* é maior com um valor de 5W, comparativamente ao valor com 10W, o que não faz qualquer sentido.

Posteriormente, decidimos verificar a correlação entre as duas variáveis que estão a ser testadas.

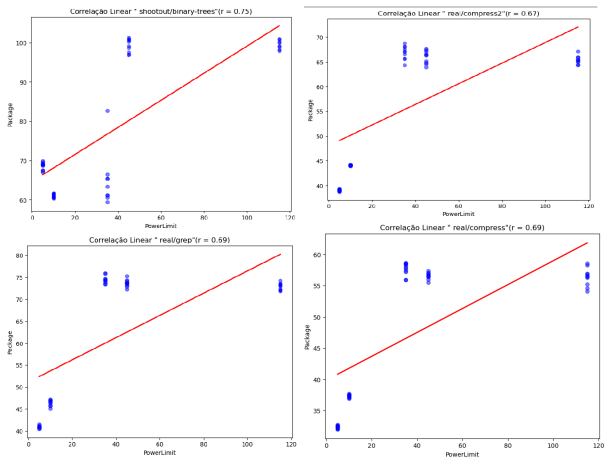

Figure 14: Correlação para *benchmarks* de *haskell* a comparar *Package* e *PowerLimit*

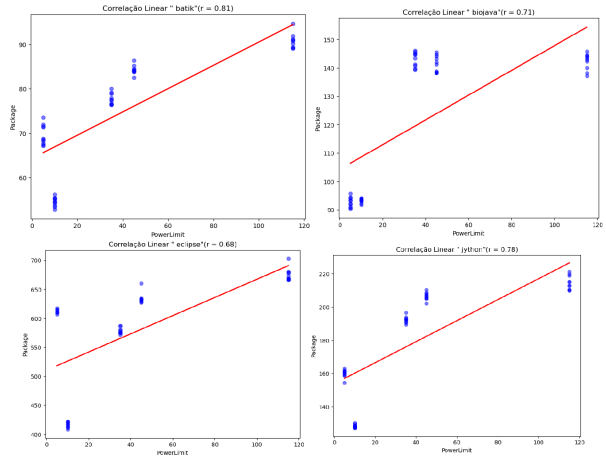

Figure 15: Correlação para *benchmarks* de *java* a comparar *Package* e *PowerLimit*

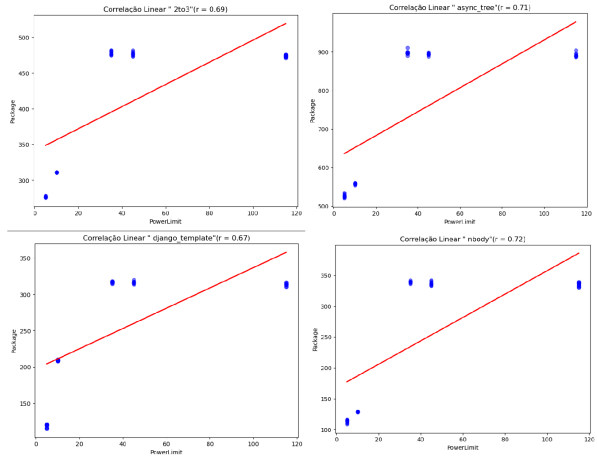

Figure 16: Correlação para *benchmarks* de *python* a comparar *Package* e *PowerLimit*

Como podemos verificar pelos gráficos com a regressão linear, à medida que o *PowerLimit* sobe, o *Package* tende a aumentar, o que coincide com a espetativa. À medida que damos mais potência ao computador, mais recursos o mesmo utiliza. Por fim calculamos a correlação de Pearson, Spearman e Kendall para todos os *benchmarks*.

	Program	Pearson	Pearson_p	Spearman	Spearman_p	Kendall	Kendall_p
0	imaginary/rfib	-0.619736	3.399128e-06	-0.762414	4.761172e-10	-0.608433	2.623789e-08
1	real/compress	-0.605655	4.009439e-06	-0.740572	1.182101e-09	-0.573197	8.287650e-08
2	real/compress2	-0.583656	1.661145e-05	-0.703957	3.386223e-08	-0.546353	5.951941e-07
3	real/gg	-0.667350	4.079684e-07	-0.859102	2.192666e-14	-0.677439	9.135221e-10
4	real/grep	-0.617411	3.787592e-06	-0.874752	9.317308e-16	-0.743782	9.721644e-12
5	real/rsa	-0.589905	3.934330e-05	-0.822290	2.426276e-11	-0.691919	2.374662e-09
6	shootout/binary-trees	-0.616114	4.021767e-06	-0.892823	3.405101e-17	-0.739418	1.248874e-11
7	shootout/fannkuch-redux	-0.623484	1.706813e-06	-0.914660	4.091015e-20	-0.792452	1.145026e-13
8	shootout/spectral-norm	-0.614130	4.406050e-06	-0.801716	1.275885e-11	-0.643448	3.800222e-09
9	spectral/sorting	-0.608822	7.142516e-06	-0.696073	7.838895e-08	-0.544260	8.567240e-07

Figure 17: Correlação de Pearson, Spearman e Kendall para *benchmarks* de *python* a comparar *Package* e *PowerLimit*

	Program	Pearson	Pearson_p	Spearman	Spearman_p	Kendall	Kendall_p
0	avrora	-0.564877	0.000035	-0.691598	7.333472e-08	-0.535088	9.671018e-07
1	batik	-0.593656	0.000009	-0.929775	1.362748e-21	-0.813336	4.828734e-14
2	biojava	-0.598448	0.000004	-0.785991	1.361058e-11	-0.633877	2.013618e-09
3	eclipse	-0.571033	0.000042	-0.939537	1.247490e-21	-0.833493	8.614463e-14
4	graphchi	-0.601259	0.000008	-0.759291	6.160737e-10	-0.598768	4.193553e-08
5	jme	-0.599461	0.000004	-0.863477	7.122610e-16	-0.734245	3.967861e-12
6	jython	-0.597351	0.000005	-0.947652	1.919851e-25	-0.843876	1.355685e-15
7	kafka	-0.559774	0.000029	-0.838503	5.535895e-14	-0.700815	5.400638e-11
8	spring	-0.611959	0.000005	-0.816168	2.735913e-12	-0.707269	9.065535e-11
9	tomcat	-0.605464	0.000005	-0.957438	1.830671e-26	-0.863842	1.222109e-15

Figure 18: Correlação de Pearson, Spearman e Kendall para *benchmarks* de *python* a comparar *Package* e *PowerLimit*

	Program	Pearson	Pearson_p	Spearman	Spearman_p	Kendall	Kendall_p
0	2to3	-0.616118	4.021078e-06	-0.582304	1.756540e-05	-0.405552	2.084105e-04
1	async_tree	-0.595957	6.245252e-06	-0.700943	2.044407e-08	-0.520680	1.086152e-06
2	chameleon	0.275636	7.014252e-02	-0.103298	5.046041e-01	-0.005855	9.587026e-01
3	django_template	-0.698508	2.399410e-08	-0.715324	7.679740e-09	-0.540913	4.174748e-07
4	docutils	-0.563111	3.778848e-05	-0.711272	2.103589e-08	-0.536641	8.804632e-07
5	html5lib	-0.195458	1.830698e-01	-0.008254	9.555986e-01	-0.018649	8.628594e-01
6	json_dumps	-0.711190	4.373119e-08	-0.655419	1.029029e-06	-0.464190	3.270944e-05
7	nbody	-0.302152	3.899960e-02	-0.193715	1.920014e-01	-0.068648	5.295613e-01
8	pidigits	-0.616709	1.063978e-05	-0.698535	1.903500e-07	-0.529961	3.894196e-06
9	tornado_http	-0.401643	3.839756e-03	-0.048020	7.405291e-01	0.095772	3.645534e-01

Figure 19: Correlação de Pearson, Spearman e Kendall para *benchmarks* de *python* a comparar *Package* e *PowerLimit*

Como todos os valores de "p" representados nas imagens acima são inferiores a 0.05, significa que a correlação entre as duas variáveis é estatisticamente significante.

Após toda a análise feita a equipa pode concluir que o uso de *PowerCap* influencia os recursos utilizados pelo computador.

C. *Correlação entre o PowerCap e o tempo de execução*

A próxima etapa do trabalho pede-nos para provar, caso exista, a correlação entre as variáveis *PowerLimit* e *Time*.

Similarmente à análise feita na fase anterior começamos por fazer um *Scatter Plot* para termos um visão geral dos nossos dados.

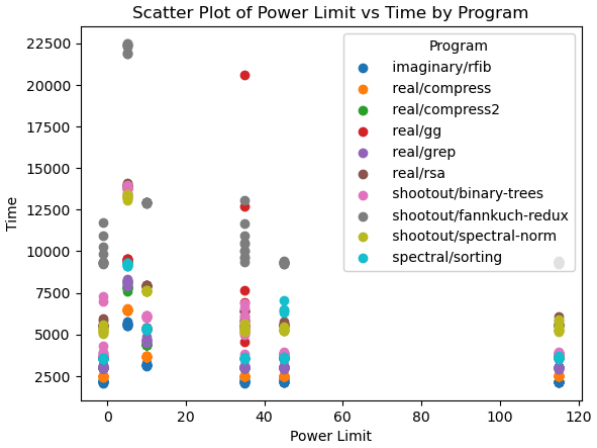

Figure 20: *Scatter Plot* para todos os *benchmarks* de *haskell* a comparar os valores *Time* e *PowerLimit*

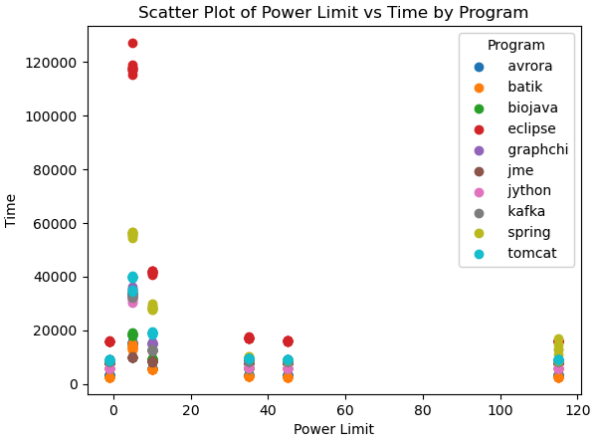

Figure 21: *Scatter Plot* para todos os *benchmarks* de *java* a comparar os valores *Time* e *PowerLimit*

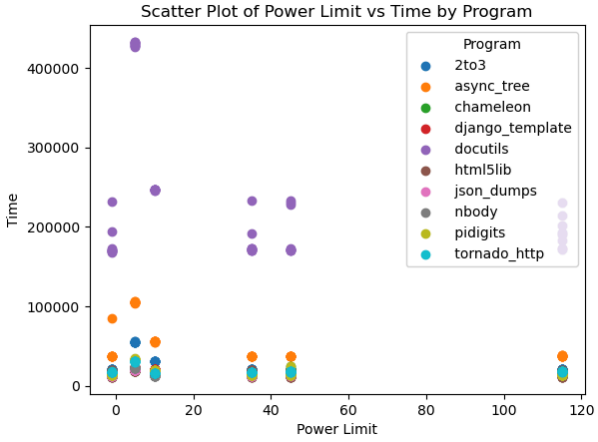

Figure 22: *Scatter Plot* para todos os *benchmarks* de *python* a comparar os valores *Time* e *PowerLimit*

A partir da visualização dos *Scatter Plots* podemos observar que em valores mais baixos do *PowerLimit* o tempo de execução do programa tende a aumentar, o que faz sentido. Quanto menos potência o computador tiver para executar uma determinada tarefa, mais tempo demora a conclui-la. No entanto, mais uma vez, não podemos tirar estas conclusões simplesmente com base nestes gráficos.

A seguir, a equipa prosseguiu com a análise dos *Box Plots* de todos os *benchmarks*:

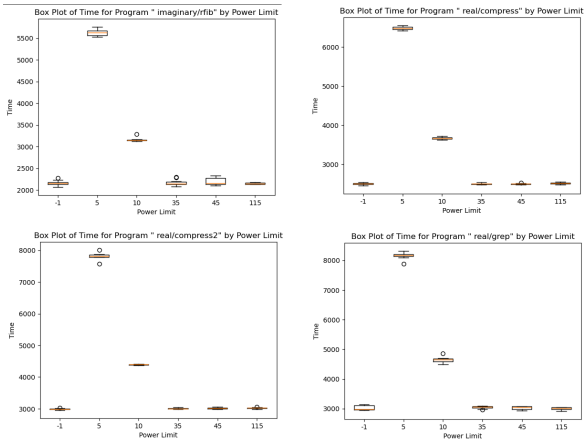

Figure 23: *Box Plot* para alguns dos *benchmarks* de *haskell*, a comparar os valores *Time* e *PowerLimit*

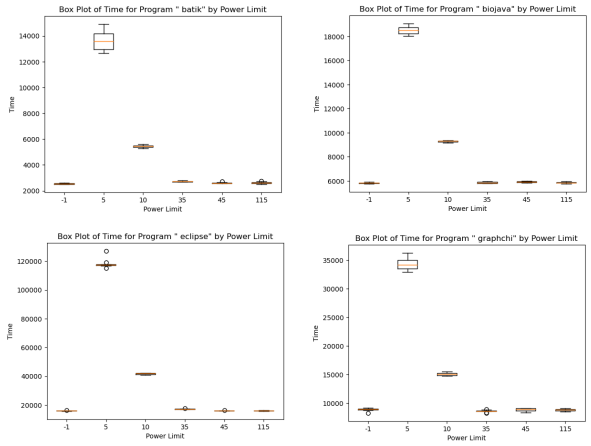

Figure 24: *Box Plot* para alguns dos *benchmarks* de *java*, a comparar os valores *Time* e *PowerLimit*

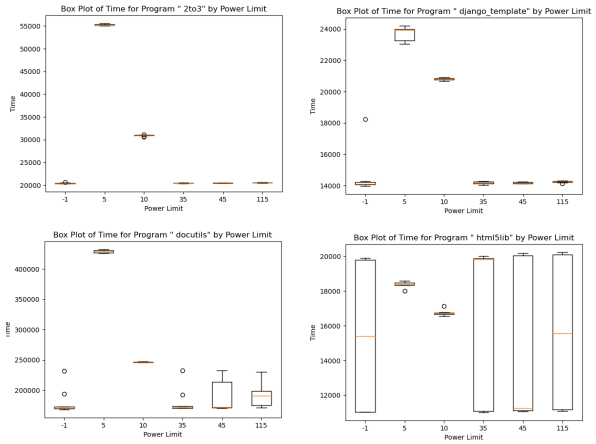

Figure 25: *Box Plot* para alguns dos *benchmarks* de *python*, a comparar os valores *Time* e *PowerLimit*

Todos os gráficos permitem-nos concluir a mesma coisa. Vemos um aumento consideravelmente grande do valor da coluna *Time* quando o *PowerLimit* é 5. Depois desce parcialmente quando tem valor de 10W, e estagna mais uma vez quando chega aos 35W.

Por fim, analisamos a correlação entre as duas variáveis que estão a ser estudadas, calculando também a correlação de Pearson, Spearman e Kendall.

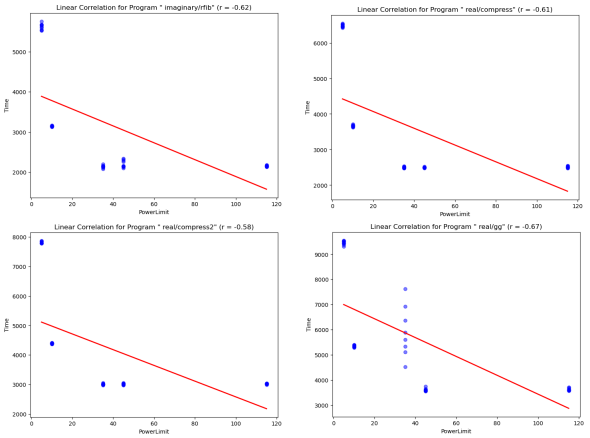

Figure 26: Correlação para *benchmarks* de *haskell* a comparar *Time* e *PowerLimit*

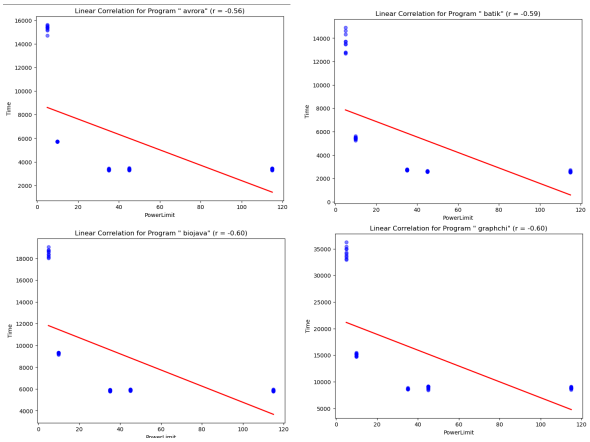

Figure 27: Correlação para *benchmarks* de *java* a comparar *Time* e *PowerLimit*

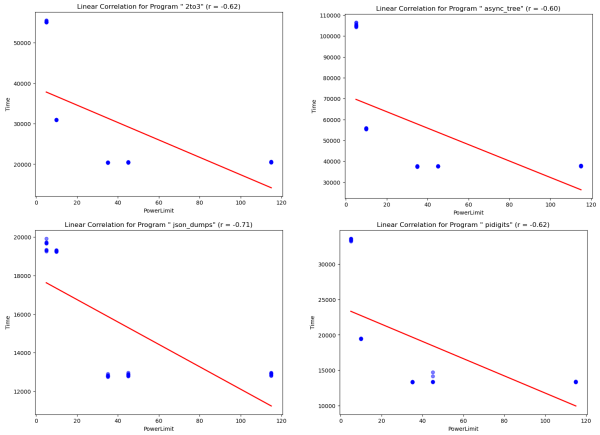

Figure 28: Correlação para *benchmarks* de *python* a comparar *Time* e *PowerLimit*

	Program	Pearson	Pearson_p	Spearman	Spearman_p	\
0	imaginary/rfib	-0.619736	3.399128e-06	-0.762414	4.761172e-10	
1	real/compress	-0.605655	4.009439e-06	-0.740572	1.182101e-09	
2	real/compress2	-0.583656	1.661145e-05	-0.703957	3.386223e-08	
3	real/gg	-0.667350	4.079684e-07	-0.859102	2.192666e-14	
4	real/grep	-0.617411	3.787592e-06	-0.874752	9.317308e-16	
5	real/rsa	-0.589905	3.934330e-05	-0.822290	2.426276e-11	
6	shootout/binary-trees	-0.616114	4.021767e-06	-0.892823	3.405101e-17	
7	shootout/fannkuch-redux	-0.623484	1.706813e-06	-0.914660	4.091015e-20	
8	shootout/spectral-norm	-0.614130	4.406050e-06	-0.801716	1.275885e-11	
9	spectral/sorting	-0.608822	7.142516e-06	-0.696073	7.838895e-08	

	Kendall	Kendall_p
0	-0.608433	2.623789e-08
1	-0.573197	8.287650e-08
2	-0.546353	5.951941e-07
3	-0.677439	9.135221e-10
4	-0.743782	9.721644e-12
5	-0.691919	2.374662e-09
6	-0.739418	1.248874e-11
7	-0.792452	1.145026e-13
8	-0.643448	3.800222e-09
9	-0.544260	8.567240e-07

Figure 29: Correlação de Pearson, Spearman e Kendall para *benchmarks* de *haskell* a comparar *Time* e *PowerLimit*

	Program	Pearson	Pearson_p	Spearman	Spearman_p	Kendall	\
0	avrora	-0.564877	0.000035	-0.691598	7.333472e-08	-0.535088	
1	batik	-0.593656	0.000009	-0.929775	1.362748e-21	-0.813336	
2	biojava	-0.598448	0.000004	-0.785991	1.361058e-11	-0.633877	
3	eclipse	-0.571033	0.000042	-0.939537	1.247490e-21	-0.833493	
4	graphchi	-0.601259	0.000008	-0.759291	6.160737e-10	-0.598768	
5	jme	-0.599461	0.000004	-0.863477	7.122610e-16	-0.734245	
6	jython	-0.597351	0.000005	-0.947652	1.919851e-25	-0.843876	
7	kafka	-0.559774	0.000029	-0.838503	5.535895e-14	-0.700815	
8	spring	-0.611959	0.000005	-0.816168	2.735913e-12	-0.707269	
9	tomcat	-0.605464	0.000005	-0.957438	1.830671e-26	-0.863842	

	Kendall_p
0	9.671018e-07
1	4.828734e-14
2	2.013618e-09
3	8.614463e-14
4	4.193553e-08
5	3.967861e-12
6	1.355685e-15
7	5.400638e-11
8	9.065535e-11
9	1.222109e-15

Figure 30: Correlação de Pearson, Spearman e Kendall para *benchmarks* de *java* a comparar *Time* e *PowerLimit*

	Program	Pearson	Pearson_p	Spearman	Spearman_p	Kendall	\
0	2to3	-0.616118	4.021078e-06	-0.582304	1.756540e-05	-0.405552	
1	async_tree	-0.595957	6.245252e-06	-0.700943	2.044407e-08	-0.520680	
2	chameleon	0.275636	7.014252e-02	-0.103298	5.046041e-01	-0.005855	
3	django_template	-0.698508	2.399410e-08	-0.715324	7.679740e-09	-0.540913	
4	docutils	-0.560357	3.445283e-05	-0.699683	3.141266e-08	-0.526158	
5	html5lib	-0.195458	1.830698e-01	-0.008254	9.555986e-01	-0.018649	
6	json_dumps	-0.711190	4.373119e-08	-0.655419	1.029029e-06	-0.464190	
7	nbody	-0.302152	3.899960e-02	-0.193715	1.920014e-01	-0.068648	
8	pidigits	-0.616709	1.063978e-05	-0.698535	1.903500e-07	-0.529961	
9	tornado_http	-0.401643	3.839756e-03	-0.048020	7.405291e-01	0.095772	

	Kendall_p
0	2.084105e-04
1	1.086152e-06
2	9.587026e-01
3	4.174748e-07
4	1.093221e-06
5	8.628594e-01
6	3.270944e-05
7	5.295613e-01
8	3.894196e-06
9	3.645534e-01

Figure 31: Correlação de Pearson, Spearman e Kendall para *benchmarks* de *python* a comparar *Time* e *PowerLimit*

Como podemos ver pelos gráficos e pelos valores de "p" que são menores de 0.05, a relação entre as colunas *Time* e *PowerLimit* é estatisticamente significante.

D. ***Comparar o desempenho entre diferentes linguagens***

Para analisar o desempenho entre as diferentes linguagens, podemos utilizar os dados obtidos em diferentes níveis de Powercap, estabelecendo relações entre os tempos de execução e a energia consumida. A hipótese inicial do grupo é que a linguagem que apresentar os maiores ganhos de performance com variações no consumo de energia será considerada a de melhor desempenho.

Nas alíneas anteriores, foi analisado o impacto do Powercap no consumo de energia e no tempo de execução, constituindo o ponto de partida para esta análise.

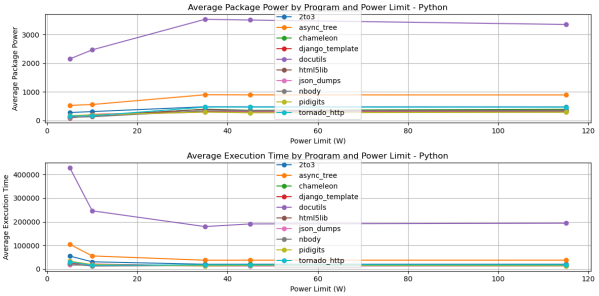

Figure 32: Python Package and Execution Time per PowerLimit

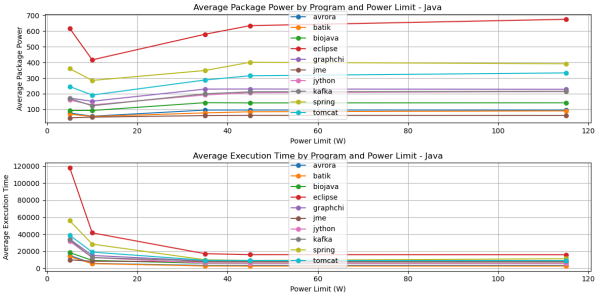

Figure 33: Java Package and Execution Time per PowerLimit

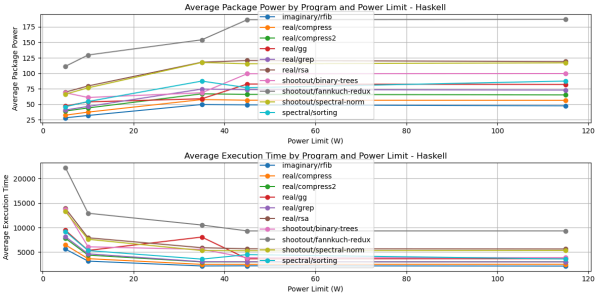

Figure 34: Haskell Package and Execution Time per PowerLimit

Após uma vizualização inicial dos valores de energia e tempo de execução em diferentes PowerLimits, o grupo fez uma preparação dos dados básica :

- **Removendo Colunas Vazias** : coluna "DRAM" e "GPU".

- **Eliminando Linhas Inválidas** : valores anómalos como valores de tempo menores que 0.
- **Removendo Outliers** : removendo valores para além dos intervalos $1.5 * Q1$ e $1.5 * Q3$.

De modo a visualizar a relação entre Package e Execution Time, traçou-se os seus os valores num mesmo gráfico para todos os benchmarks das três linguagens em analise. Uma conclusão comum é que com o aumento do valor de Powercap o tempo de execução diminui, tal que apartir de um certo valor não representa mais nenhum ganho significativo.

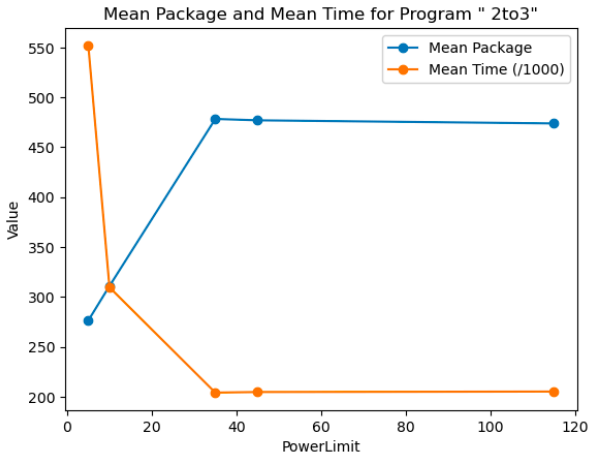

Figure 35: Package and Time in Python Benchmark “2to3”

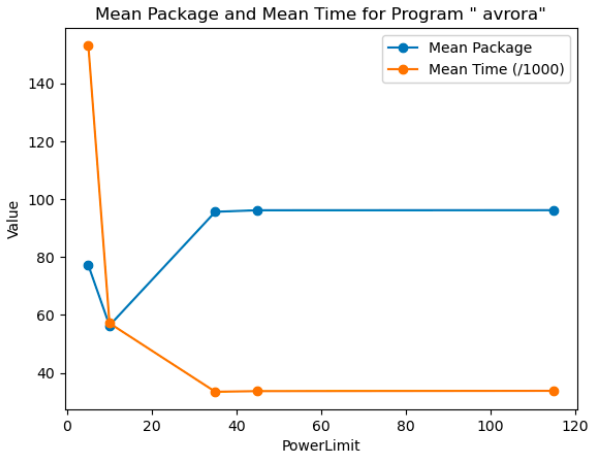

Figure 36: Package and Time in Java Benchmark “avrora”

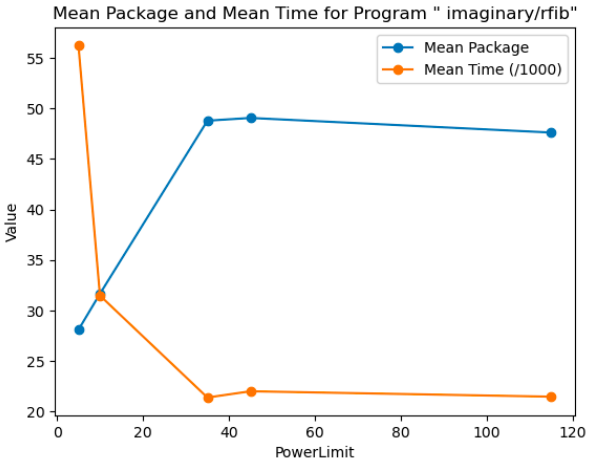

Figure 37: Package and Time in Haskell Benchmark “imaginary/rfib”

No entanto esta análise superficial não é suficiente para se concluir qual a melhor linguagem em termos de desempenho e qual tira mais proveito do Powercap. Para isso a equipa calculou, para todos os benchmarks, o coeficiente de correlação entre Time e Package representado pela a sua linha de regressão linear. Como vemos nos gráficos seguintes.

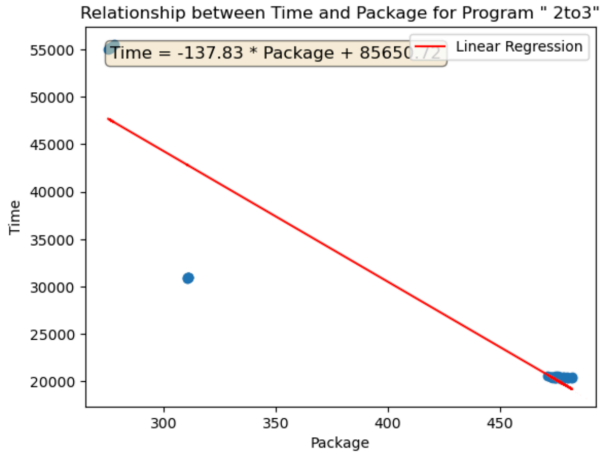

Figure 38: Package and Time Correlation in Python Benchmark “2to3”

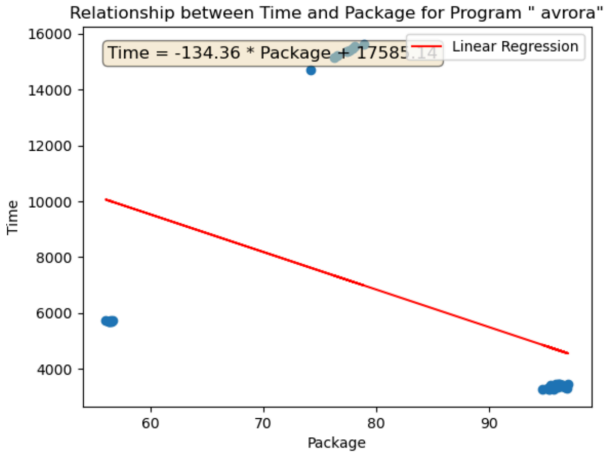

Figure 39: Package and Time Correlation in Java Benchmark “avrora”

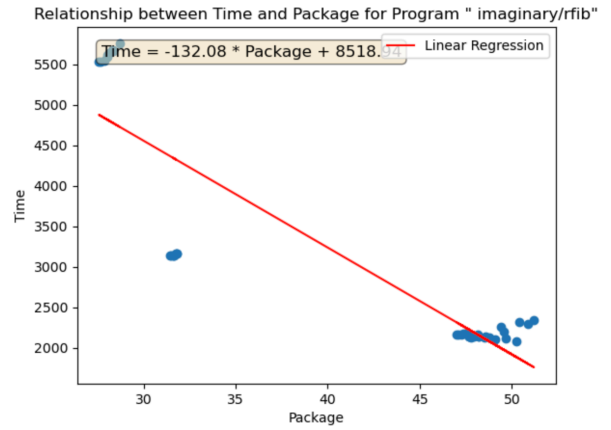

Figure 40: Package and Time Correlation in Haskell Benchmark “imaginary/rfib”

A formula que representa a linha de regressão é, $\text{Time} = \text{slope} * \text{Package} + \text{intercept}$, onde o valor de *slope* representa o ritmo de descrescimento do tempo de execução por unidade de Package e *intercept* o desvio inicial do tempo.

Sendo assim a equipa reuniu os valores de todos os benchamarks por linguagem e chegou a um valor médio de *slope* e de *intercept*.

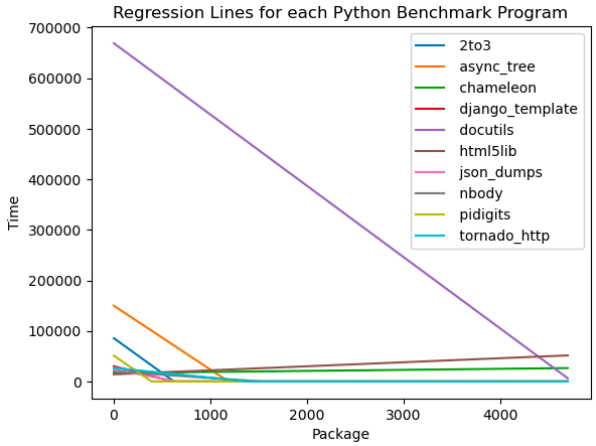

Figure 41: Regression Lines in all Python Benchmarks

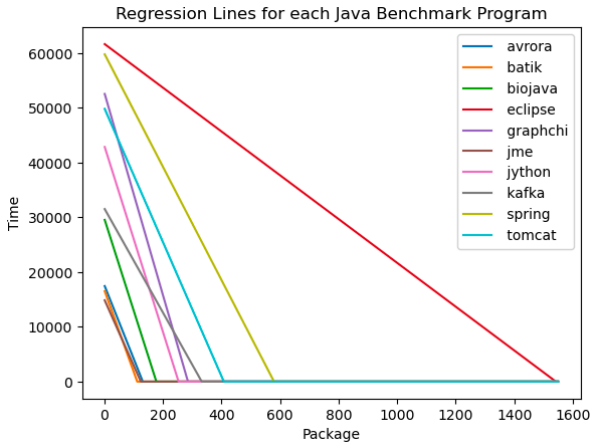

Figure 42: Regression Lines in all Java Benchmarks

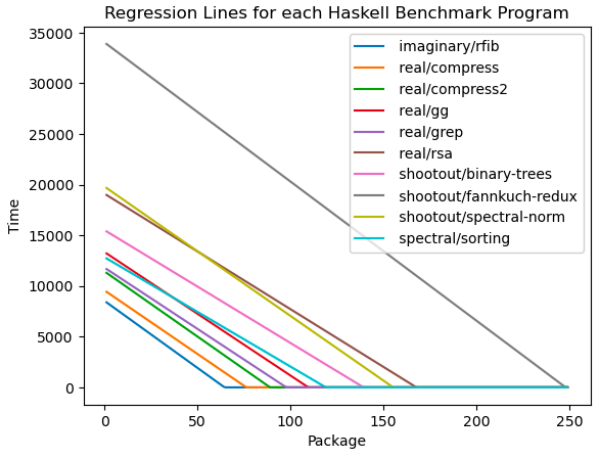

Figure 43: Regression Lines in all Haskell Benchmarks

Para chegarmos à conclusão de qual a linguagem com melhor desempenho e em qual se tira melhor partido do Powercap, temos de observar qual tem o valor médio de *slope* mais alto. Pois isto significa que é nesta que teremos mais ganhos por unidade de energia.

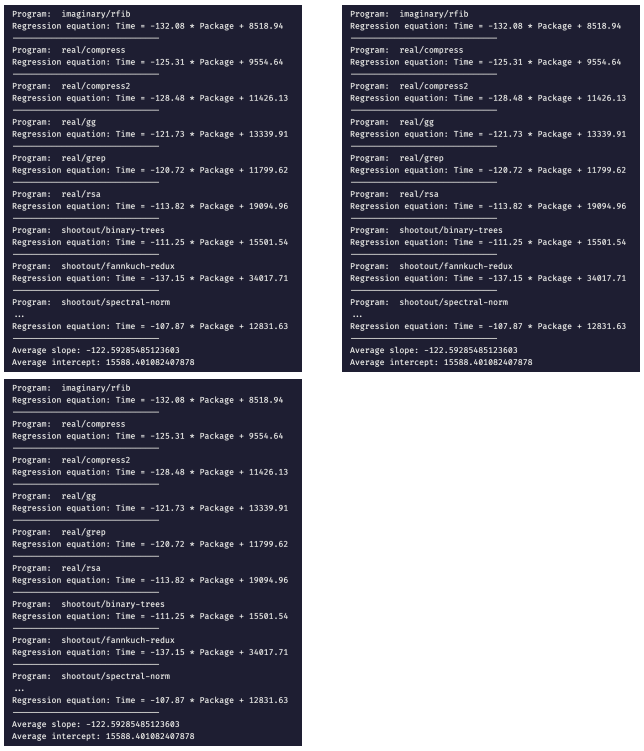

Figure 44:

Os valores obtidos são:

- Python
  - Average slope: −64.67766579797102
  - Average intercept: 108781.78929216231
- Java
  - Average slope: −128.69811153642993
  - Average intercept: 37785.97711352551
- Haskell
  - Average slope: −122.59285485123603
  - Average intercept: 15588.401082407878

O Python apresenta o pior desempenho, pois tem um valor médio de inclinação (slope) baixo, como pode ser constatado em algumas das suas linhas de regressão linear. O Java e Haskell demonstram bom ganho de desempenho por unidade de energia. O Java, em particular, possui os benchmarks mais pesados, evidenciado pelo valor mais alto de interceptação (intercept). No entanto, Java é ligeiramente melhor que Haskell, pois apresenta uma inclinação (slope) ligeiramente mais alta, logo um melhor ganho.

Em termos de tirar proveito do *Powercap* o grupo constata que o Java e Haskell tem bastantes ganhos por unidade de energia, o que faz com que com um limite razoável seja possível obter um tempo de execução satisfazível.

V. Conclusão

Chegando ao fim deste trabalho prático, o grupo ficou um pouco desapontado dado que não tivemos resultados animadores na primeira fase, e acabamos por ficar com dados impossíveis de serem analisados sem chegar a uma conclusão do porquê.

No entanto, mesmo utilizando os dados de outro grupo, consideramos que fizemos um bom trabalho, demonstrando de forma eficaz e sucinta o efeito do uso do *PowerCap* em diversos *benchmarks* de variadas linguagens.

Para além disso, os resultados permitiram-nos observar variações significativas no consumo de energia entre as diferentes linguagens, refletindo as suas a eficiência dos compiladores ou interpretadores utilizados. Linguagens de baixo nível, como *Haskell*, geralmente apresentaram um consumo energético menor e uma maior eficiência de execução, enquanto que linguagens de alto nível, como *Python*, mostraram um consumo energético mais elevado devido à sobrecarga do interpretador e à gestão dinâmica de recursos.

Em suma, o uso do RAPL forneceu dados importantes sobre o consumo de energia/tempo dos programas consoante o uso do *PowerCap*, e mostra uma área a que devemos estar atentos, sobre programação mais verde e sustentável.

References

[1] R. P. M. C. F. R. R. R. J. C. João Paulo Fernandes João Saraiva, “Energy Efficiency across Programming Languages - How Do Energy, Time, and Memory Relate?.” [Online]. Available: https://www.researchgate.net/publication/320436353_Energy_efficiency_across_programming_languages_how_do_energy_time_and_memory_relate

[2] B. G. Sebastian Graf, “nofib.” [Online]. Available: https://gitlab.haskell.org/ghc/nofib

[3] “Benchmarks.” [Online]. Available: https://dacapobench.sourceforge.net/benchmarks.html

[4] “Apache Kafka.” [Online]. Available: https://kafka.apache.org/

[5] “About Biojava.” [Online]. Available: https://biojava.org/wiki/About

[6] “GraphChi - disk-based large-scale graph computation.” [Online]. Available: https://github.com/GraphChi/graphchi-cpp

[7] “Jme.” [Online]. Available: https://jmonkeyengine.org/

[8] C. H. A. K. K. M. R. B. A. D. D. F. D. F. S. G. M. H. A. H. M. J. H. L. J. E. B. M. A. P. D. S. T. V. D. V. D. B. W. Stephen Blackburn Robin Garner, “The DaCapo Benchmarks:Java Benchmarking Development and Analysis,” 2006.

[9] “Tornado.” [Online]. Available: https://www.tornadoweb.org/en/stable/

[10] “Benchmarks.” [Online]. Available: https://pyperformance.readthedocs.io/benchmarks.html#available-benchmarks

[11] “RaplCap.” [Online]. Available: https://github.com/powercap/raplcap

[12] “Intel Core i7-1065G7 Processor.” [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/196597/intel-core-i7-1065g7-processor-8m-cache-up-to-3-90-ghz.html

[13] “Intel Core i7-4720HQ Processor.” [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/78934/intel-core-i74720hq-processor-6m-cache-up-to-3-60-ghz/specifications.html

[14] “Intel Core i7-4720HQ Processor.” [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/78934/intel-core-i74720hq-processor-6m-cache-up-to-3-60-ghz/specifications.html